Lecture 14

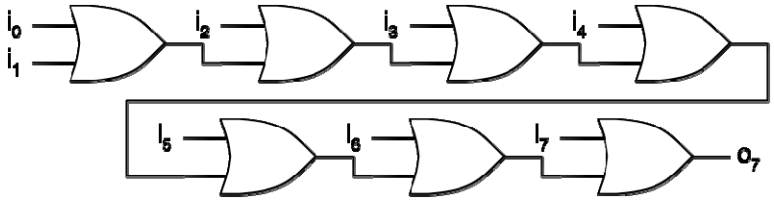
- Logistics
  - Midterm 1: ave 88, median 89, std 9. good job!
  - HW4 due on Wednesday
  - Lab5 is going on this week
- Last lecture
  - Multi-level logic
  - Timing diagrams
- Today
  - Time/space trade offs: Parallel prefix trees
  - Adders
  - The conclusion of combinational logic!!!

The "WHY" slide

- Timing/space trade offs
  - In real life, complex logic circuits you will work on will not have one minimum circuit. You will have to learn to understand what parameters to optimize your design on, and be able to come up with "trade offs" suitable for your application or customer's needs.
- Adders
  - Arithmetic logic units (ALUs) such as adders and multipliers perform most computer instructions. Therefore, it is critical to know how it works, how it scales, and how it may be optimized for time/space.

What do we mean by time/speed tradeoff?: Linear chains vs. trees

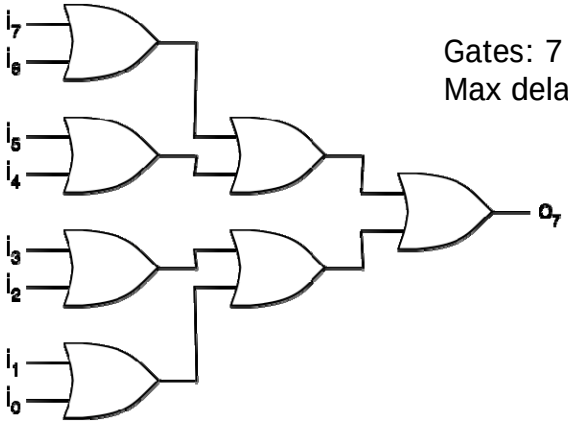
- Lets say we want to implement an 8-input OR function with only 2-input gates



Gates: 7
Max delay: 7

Linear chains vs. trees

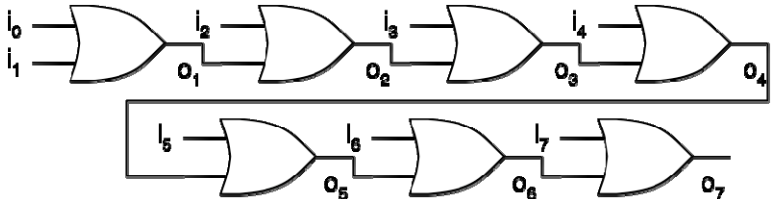
- Now consider the tree version



Gates: 7
Max delay: 3

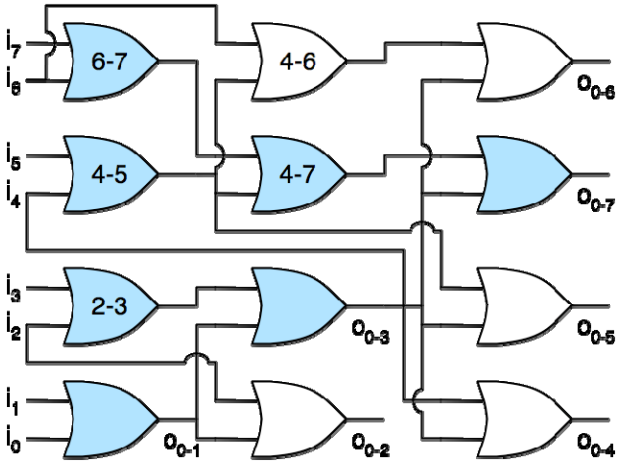
And now we change the problem slightly

- Build a circuit that takes 8 single bit inputs and calculates the OR of the first 2, the OR of the first 3, the OR of the first 4, and so on, up to the OR of all 8



Gates: 7
Max delay: 7

The tree version of the prefix circuit

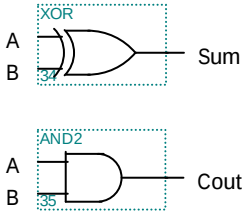


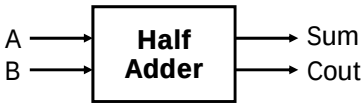
Gates: 12
Max delay: 3

Binary half adder

- 1-bit half adder
  - Computes sum, carry-out
    - No carry-in
  - Sum = A'B + AB' = A xor B
  - Cout = AB

A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

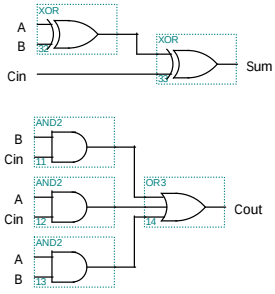


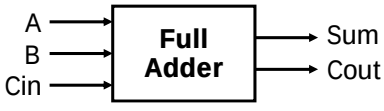


Binary full adder

- 1-bit full adder
  - Computes sum, carry-out
    - Carry-in allows cascaded adders
  - Sum = Cin xor A xor B
  - Cout = ACin + BCin + AB

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





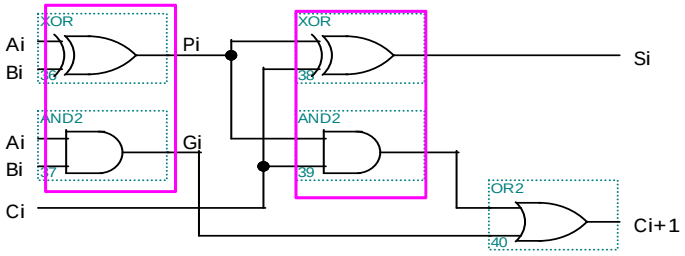
Full adder: using 2 half adders

- Multilevel logic
  - Slower
  - Fewer gates
    - 2 XORs, 2 ANDs, 1 OR

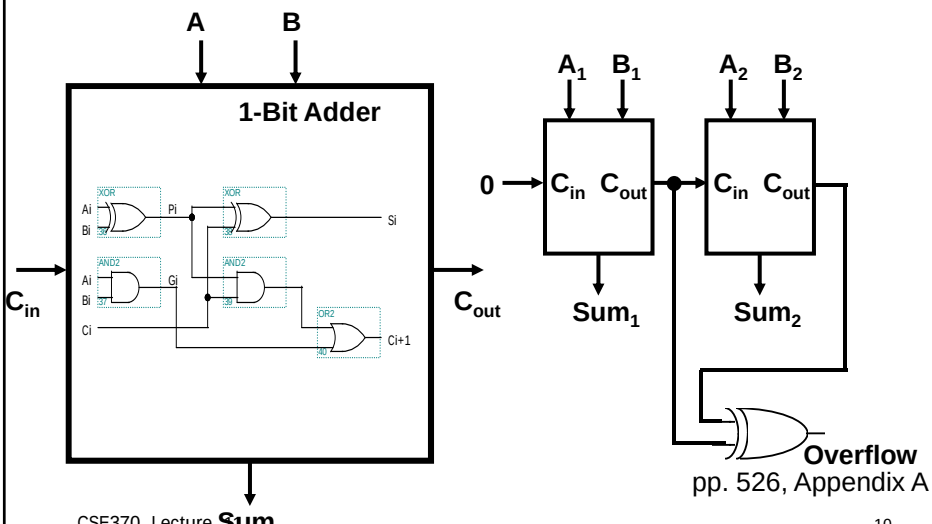
Sum = (A ⊕ B) ⊕ Cin

Cout = ACin + BCin + AB
= (A ⊕ B)Cin + AB

A	B	C_{in}	S	C_{out}	C_{out}
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

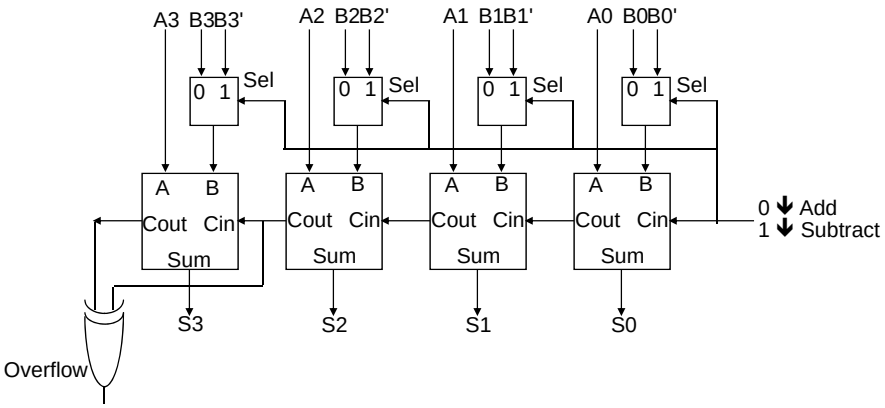


2-bit ripple-carry adder



pp. 526, Appendix A

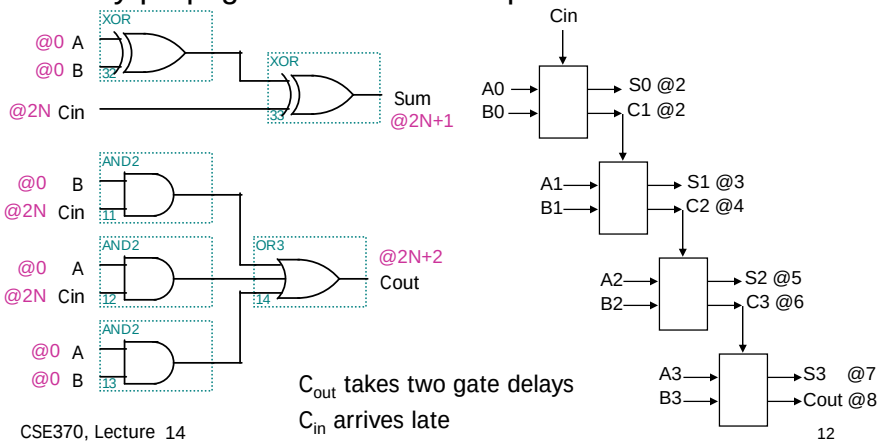
4-bit ripple-carry adder



- Turns out it is easy to convert to subtractor
  - 2s complement: A – B = A + (–B) = A + B' + 1

Let's talk about speed optimization

- Back to 2-level structure for speed
- Carry propagation limits adder speed

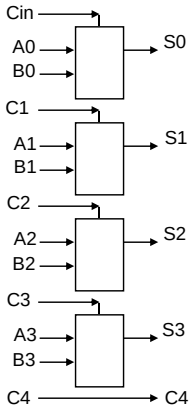


C_{out} takes two gate delays

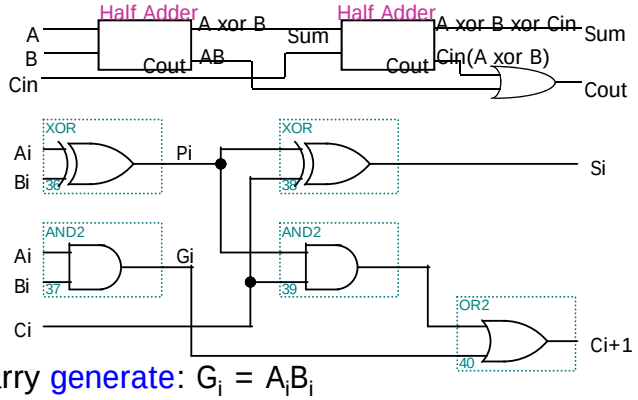
C_{in} arrives late

Can we be clever and speed this up?

- Let's Compute all the carries in parallel
  - Derive carries from the data inputs
    - Not from intermediate carries
    - Use two-level logic
  - Compute all sums in parallel
  - How do we do that???



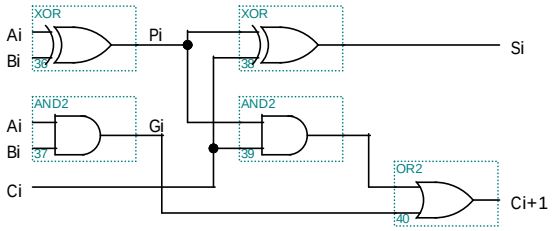
Solution: Create a carry lookahead logic Step 1: Getting Pi an Gi



- Carry generate: $G_i = A_iB_i$
  - Generate carry when A = B = 1
- Carry propagate: $P_i = A_i$ xor B_i
  - Propagate carry-in to carry-out when (A xor B) = 1

Solution: Carry-lookahead logic
Step 2: Calculate Sum and Cout

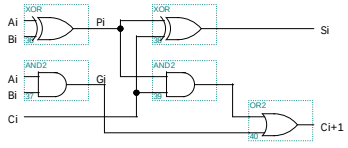
- Sum and Cout in terms of generate/propagate:
  - $S_i = A_i \text{ xor } B_i \text{ xor } C_i$
    $= P_i \text{ xor } C_i$
  - $C_{i+1} = A_iB_i + C_i(A_i \text{ xor } B_i)$
    $= G_i + C_iP_i$



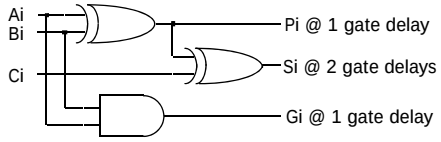
Solution: Carry-lookahead logic
Step 3: Express all carry in terms of C0

- Re-express the carry logic in terms of G and P
  $C_1 = G_0 + P_0C_0$
  $C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$
  $C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
  $C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

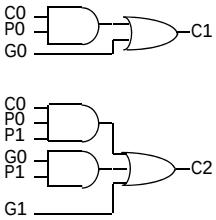
- Implement each carry equation with two-level logic
  - Derive intermediate results directly from inputs
    - Rather than from carries
  - Allows "sum" computations to proceed in parallel

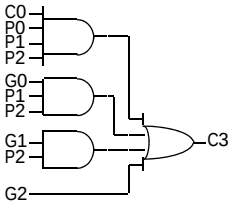


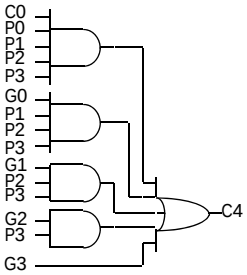
Solution: carry-lookahead logic
Step 4: implement with 2-level logic



Logic complexity increases with adder size







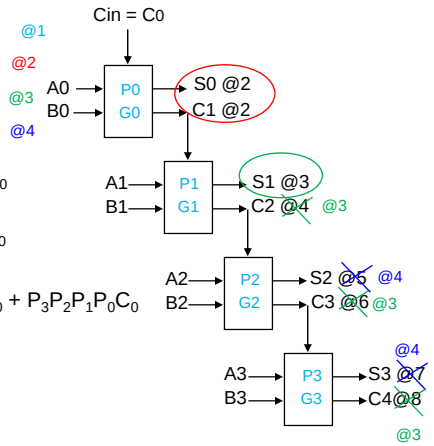
With Carry lookahead logic

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2$
$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$C_4 = G_3 + P_3C_3$
$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$



Carry lookahead logic summary

- Compute all the carries in parallel
  - Derive carries from the data inputs
    - Not from intermediate carries
    - Use two-level logic
  - Compute all sums in parallel
- Cascade simple adders to make large adders
- Speed improvement
- Complex combinational logic